TotalView Creating Type Transformations

Etnus

April 2003

Version 6.2

Contents

TTF Overview 1

The Type Transformation Facility (TTF) lets you define the way TotalView displays aggregate data. *Aggregate data* is simply a collection of data elements. These elements can even be other aggregated elements. In most cases, you will be creating transformations that model data that your program stores in an array-like or list-like way. You can also transform arrays of structures.

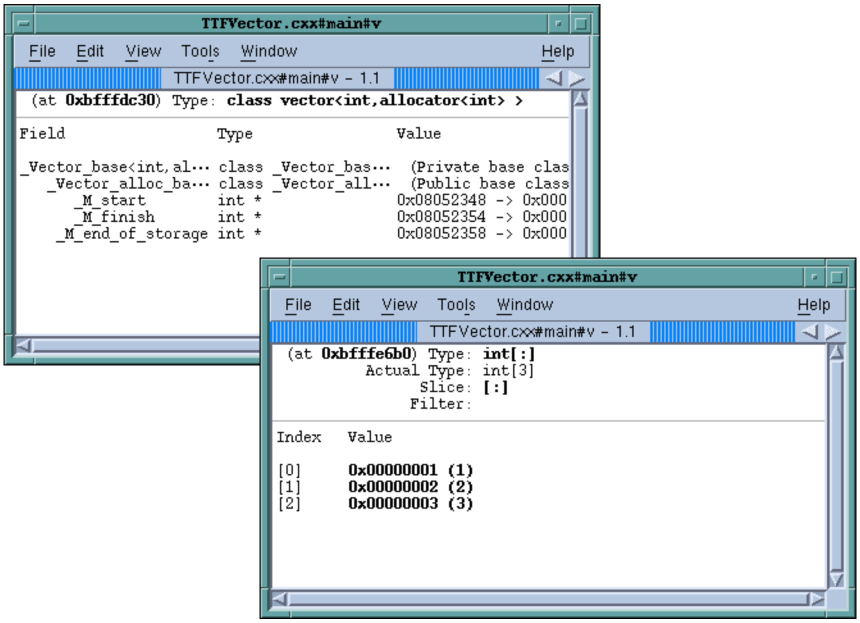
This chapter describes the TTF. It presents information on the existing transformations and an overview of how you create your own.

While Etnus supports the transformation scripts that it provides and supports the type transformation facility, we do not offer support for problems you may encounter when writing your own transformations. As you will see, writing a transformation means grappling with the way your compiler stores information and the way in which TotalView stores debugging information. Consequently, creating a type transformation is often a laborious, trial and error, iterative activity.

Why Type Transformations

Modern programming languages allow you to use abstractions such as lists, maps, and vectors to model the data that your program uses. For example, the STL (Standard Template Library) allows you to create vectors of the data contained within a class. These abstractions simplify the way in which you think of and manipulate program's data. While these abstractions simplify the way in which you can manipulate this data, they greatly complicate debugging this data when problems occur. For example, Figure 1 shows a vector transformation.

FIGURE 1: **A Vector Transformation**



The upper left window shows untransformed information. TotalView is treating this GNU C++ STL instantiation in the same way as any other class. That is, it shows the complete structure of the information, which means you are seeing the data as your compiler stored it.

While you understand the logical model that is the reason for using an STL vector, neither TotalView nor your compiler has this information. This is where type transformations come in. They give TotalView knowledge of how the data is structured and how it can access data elements.

Using Type Transformations

When TotalView begins executing, it loads its built in transformations. To locate the directory in which these files are stored, use the following CLI command:

```
dset TOTALVIEW_TCLLIB_PATH
```

Type transformations are always loaded. By default, they are turned on. From the GUI, you can control whether transformations are turned on or off by going to the **Options** Page of the **File > Preferences** Dialog Box and changing the **View simplified STL containers (and user-defined transformations)** item. For example, the following turns on type transformations:

```
dset TV::ttf true
```

Instantiating Transformations

TotalView's built-in type transformations and the transformations that you will write are CLI Tcl callback procedures. While they do other things, most callback routines tell TotalView where in memory it will find information. These definitions are called *addressing expressions*. Creating expressions and callback routines is discussed in Chapter 2, "Creating Vector Transformations," on page 13.

All callbacks need to be installed as part of a transformation. This is a two-step process:

- Use the **TV::type_transformation** command to obtain a handle that TotalView will use to identify a transformation.
- Use the **TV::type_transformation** command to associate callbacks with this handle.

Here's an example:

```
set ttf_id [TV::type_transformation create Array]

TV::type_transformation set $ttf_id \
    name             {^(class|struct) (std::)?vector *<.*>$} \
    language                            C++ \
    type_transformation_description "GNU Vector"\
    validate_callback                   vector_validate \
    type_callback                       vector_type \
    lower_bounds_callback               vector_lower_bounds \
    upper_bounds_callback               vector_extent \
    addressing_callback                 vector_addressing
```

Note The STL transformations that Etnus supplies are automatically installed when TotalView starts executing.

The first **type_transformation** command also tells TotalView that you are creating an array-like transformation. The kinds of transformations that you can create are:

- **Array**: information is laid out sequentially in memory. For example, an STL vector is an array-like organization of information.
- **List**: information is linked using pointers. For example, an STL list uses this type.
- **Map**: only used for STL maps.
- **Struct**: information is a structure whose appearance the transformation is altering.

These options are not case sensitive.

The second **type_transformation** command either provides general information or names the callback procedures. The first five elements (**name**, **language**, **validate_callback**,

type_transformation_description, and **type_callback**) are used with all transformations. Each kind of transformation such as an array or a list has additional, unique callbacks. Here, for example, is the general pattern for a **List** transformation:

```
set ttf_id [TV::type_transformation create List]

TV::type_transformation set $ttf_id \
    name                 {^(class|struct) (std::)?List *<.*>$} \
    language                         C++ \
    type_transformation_description "GNU List"\
    validate_callback                list_validate \
    type_callback                    list_type \
    list_head_addressing_callback    list_head_addressing \
    list_first_element_addressing_callback \
                               list_first_element_addressing \
    list_element_count_addressing_callback \
                               list_element_count_addressing \
    list_element_next_addressing_callback \
                               list_element_next_addressing \
    list_element_prev_addressing_callback \
                               list_element_prev_addressing \
    list_element_data_addressing_callback \
                               list_element_data_addressing
```

Struct transformations are much simpler, as they just use the basic callbacks and declarations:

```
set ttf_id [TV::type_transformation create Struct]

TV::type_transformation set $ttf_id \
    name                 {^(class|struct) (std::)?List *<.*>$} \
    language                           C++ \
    type_transformation_description   "Application struct"\
    validate_callback                  struct_validate \
    type_callback                      struct_redefine
```

Note For information on a Map transformation, consult the TTF files that came with this release.

While these examples show one call to **type_transformation**, each callback or property could be done separately. The only restriction is that everything must be defined before TotalView reads your program's symbol table. In addition, you can specify callbacks and properties in any order.

Quick Definitions of Callbacks and Properties

This section provides a quick definitions of the properties and callbacks instantiated with the **type_transformation** command. You'll find more information in Chapter 3, "TTF CLI Commands," on page 29.

Notice that the first four definitions describe properties. The other definitions describe callbacks.

Used by All

name Defines a regular expression that TotalView uses to identify the data types it will transform.

language Names the programming language. This is always C++.

compiler Identifies which compiler to associate with this transformation.

type_transformation_description
Contains a brief description of the transformation.

validate_callback
Names a procedure that checks to insure that the right data type is being transformed. Typically, it also creates and stores information used by other callback procedures.

type_callback For **Array**, **List**, and **Map** transformations, identifies the actual data type. For a **struct** transformation, this identifies the procedure that does the transforming.

Unique to Array Callbacks

lower_bounds_callback
Names a procedure that returns the addressing expression that TotalView uses to locate an array's lower bound.

upper_bounds_callback
Names a procedure that returns the addressing expression that TotalView uses when it needs to establish an array's upper bound. This allows TotalView to determine the number of elements in the array.

addressing_callback
Names a procedure that returns the addressing expression that TotalView uses to locate an array's first element.

Unique to List Callbacks

list_head_addressing_callback
Names a procedure that returns the addressing expression that locates the head of a list.

list_first_element_addressing_callback
Names a procedure that returns the addressing expression that TotalView uses to move from the head of the list to the first element in the list. TotalView appends this expression to the **list_head_addressing_callback** address expression.

list_element_count_addressing_callback
Names a procedure that returns the addressing expression that TotalView uses to get the member that contains the number of elements in the list.

list_element_next_addressing_callback
: Names a procedure that returns the addressing expression that TotalView uses to go to the next element in the list.

list_element_prev_addressing_callback
: Names a procedure that returns the addressing expression that TotalView uses to go to the previous element in the list. You do not need to use this callback if you are transforming a singly-linked lists.

list_element_data_addressing_callback
: Names a procedure that returns the addressing expression that TotalView uses to obtain the data member within a list element.

Note As the Map type is so specialized, it will not be discussed in this book. If you have need to create a map-like transformation, you will find that the comments within the map source files to be helpful.

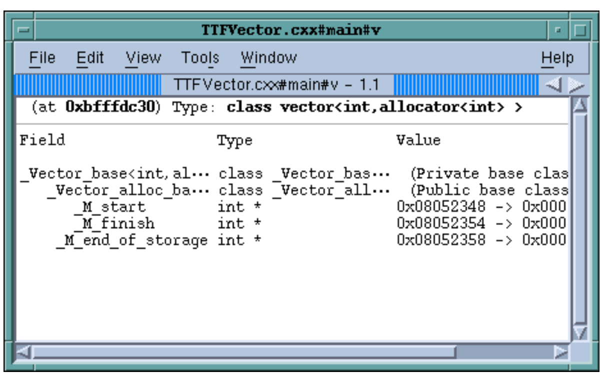
Using Addressing Expressions

Callback routines use and create addressing expressions that allow TotalView to locate where information resides. When creating these expressions, there are two issues:

- What is the structure of your information.
- How to tell TotalView how it can obtain this information.

In many cases, TotalView shows you this information. For example, here again is the structure for an STL vector:

FIGURE 2: **An STL Vector (Revisited)**



This Variable Window shows the structure of the information used by the GNU C++ compiler when it creates a vector. So, if you're going to be writing a transformation for a GNU C++ vector, your addressing expression would need to move through the class hierarchy and from one element to another. That is, you will need to tell TotalView where the data elements reside in relation to the beginning of the data structure. You'll see how this is done in the first half of Chapter 2, "Creating Vector Transformations".

Before creating these expressions, however, you'll need to know what TotalView is doing when it sees a data type that it will be transforming. Here are the steps:

1. When symbols are being read, TotalView checks to see if the symbol's data type matches any of the regular expression for registered type transformations.
2. If the symbol matches the regular expression entered into the **TV::type_transformation**'s **name** specifier, TotalView invokes that transformation's **validate_callback** procedure. It also sends the symbol's symbol ID to this procedure.
3. Your procedure will return a true or false value indicating if the symbol should be transformed. In other words, matching the regular expression indicates that the data type can be transformed. The validation routine indicates if it will be transformed.

   This routine performs two kinds of operations. The first insures that the name of the type is really what you want transformed. That is, while the data type fulfills the requirements of the regular expression, it could be similar to something you don't want transformed.

   In most cases, this validate procedure also creates addressing expressions or store data that other callback routines will use. While these other callbacks could create the addressing expressions and information they need, the operations involved in validating a data structure are similar. So doing most of the work in the validation routines just simplifies the creation of these other callback routines.

   When you go over the vector example in Chapter 2, "Creating Vector Transformations," on page 13, you'll probably think that many of the checks are redundant. If what is being transformed is a vector, then a lot of what you see isn't needed. However, these checks guard against the case of something unexpected happening.
4. If the value returned by the callback routine is true, TotalView invokes each of the registered procedures and caches the results the callback returns. When it invokes a callback, it sends the same symbol ID that was sent to the validate callback
5. Each of these procedures will return an addressing expression.

Creating a type transformation, then, means that you are defining a set of address expressions that TotalView will use when it needs to display information.

Exploring Your Data

The process of creating an address expression is usually quite involved as you must write CLI routines that step through a data structure. Fortunately, TotalView comes with a number of convenience routines that will help. These routines are also described in Chapter 2, “Creating Vector Transformations”. As you will see, they greatly simplify the process of creating the vector callback. Once you understand how these routines work, you can use them when you write your own transformations.

Unlike the kind of programming you’re used to, writing these callbacks is probably more trial-and-error and more iterative than what you are used to. For example, the vector structure has four parts. You would probably write a validate routine than walks through the first part and returns a result. After you are satisfied that is working, you’d write the second, and so on. As you are writing the validate routines, you also need to be aware of what data other callbacks require. However, on the first pass, you probably wouldn’t want to think about them. For example, the **type_callback** needs to know an element’s data type. Only after successfully creating a validation routine would you add code to the validation routine that stores the data type.

The vector example that you will read and study is misleading. It shows something that is put together correctly and where things are done in the right place. This wasn’t how it was written. Instead, it was built a piece at a time in the way just described.

The one piece of information you will need while you’re writing these routines is the data type’s symbol ID. Unfortunately, the best place to get it is from your validation routine. While this appears to be a problem, you can get around it by creating a dummy set of procedures. For example:

```
proc foo {id} {
   return true
}

proc valid {id} {
   puts “The symbol id is: $id\n”
   return false
}

set ttf_id [TV::type_transformation create Array]

TV::type_transformation set $ttf_id \
   name            {^(class|struct) (std::)d?vector *<.*>$} \
   language                        C++ \
   validate_callback               valid \
```

```
    type_transformation_description “testing”
    lower_bounds_callback           foo \
    upper_bounds_callback           foo \
    addressing_callback             foo \
    type_callback                   foo

dset TV::ttf true
```

After you use the CLI’s **source** command to read this file, TotalView prints a symbol ID in the window from which you invoked TotalView. You can now use this ID as an argument to the convenience routines.

In addition, the TTF files that come with TotalView have a great many debugging statements that display information about what is going on. You can enable and disable the display of this information by setting the **::TV::TTF::_ttf_debug** variable.

Creating Addressing Expressions

An addressing expression tells TotalView how to locate a variable, a field in a structure, or an element in an array. This expression is a string that contains one or more commands that tell TotalView how it can locate information. For example:

```
{addc 4} {indirect}
```

This expression adds 4 to the address of the data structure, and then return the value at the address pointed to by this address.

The addressing expressions that you will write are written in TotalView’s internal addressing language. This language is written as TotalView were a “stack machine”. After you create an expression, TotalView appends them too those that it has already used to reach the instance of the object with that type.

You must place all addressing expressions within braces {} and you can structure this information as lists. When generating addressing expressions, TotalView formats each opcode/operand pair as one sublist containing the expression; for example:

```
d1.<> TV::type get 1|11 struct_fields
{bit_enum 1|12 {{bitfield_index {2>>0 unsigned}}} {}}
{wide_enum 1|13 {{bitfield_index {30>>2 unsigned}}} {}}
```

TotalView ignores the list structure when it reads an addressing expression generated by user code.

Here is an explanation of the notation and abbreviations that are used in the following tables:

ACC	Accumulator or last element on the stack.
memory[*n*]	The value read from the thread address space at address *n*.
opd	A simple numeric operation; that is, a single decimal or hexadecimal (0x...) number.

stack[*n*]	The value of the *n*th element of the stack, where **stack[0]** is the top of the stack.
TOS	Top of Stack.

For opcodes without operands, all data comes from the stack.

Note There are many more operands described here than you will probably ever use. For example, the vector example in the next chapter only uses one operand from the second table and one from the third. None from the fourth are used. Table 1 contains the most oftenly used operands. However, the vector transformation only uses five of them.

TABLE 1: **Operands Without Opcodes**

Opcode	Meaning
abs	**ACC = abs (ACC)**
and	**ACC = ACC & stack**[depth-1]
div	**ACC = ACC / stack[depth**-1]
drop	Pop **ACC** and discard
dup	Push **ACC**
indirect	**ACC = memory[ACC]**
minus	**ACC = ACC - stack[depth**-1]
mod	**ACC = ACC % stack[depth**-1]
mul	**ACC = ACC * stack[depth**-1]
neg	**ACC = - ACC**
not	**ACC = ~ ACC**
or	**ACC = ACC | stack[depth**-1]
over	Push the second entry on the stack
plus	**ACC = ACC + stack**[depth-1]
rot	Rotate the top three stack entries.
shl	**ACC = ACC << stack**[depth-1]
shr	**ACC = ACC >> stack**[depth-1] **(unsigned shift)**
shra	**ACC = ACC >> stack**[depth-1] **(signed shift)**
swap	Swap top two stack entries
value	Treat ACC as number
xor	**ACC = ACC ^ stack**[depth-1]

The following table lists opcodes with operands that also use data from the stack.

TABLE 2: **Opcodes with Operands That Use the TOS (Top of Stack)**

Opcode	Meaning
addc *opd*	**ACC = ACC** + *opd*
bitfield_index *bitopd*	Load the address of the bit field whose store address is in the TOS. This must be the last opcode in an addressing expression.

Opcode	Meaning
indirect_small *opd*	Load *opd* bytes from **memory[TOS]** and zero extend.
ldnl *opd*	Load the value at address **TOS+opd**.

The **bitfield_index** opcode is more complicated and is encoded as:

size>>shift [*un*]*signed*

where:

size Is the size in bits of the field.

shift Is the shift required to justify the field at the low-significance end of the word.

This field is sign-extended if tagged as signed; otherwise, it remains unsigned.

The following opcodes push the stack. Notice that they do not use values on the stack.

TABLE 3: **Operations with Nonstack Opcodes**

Opcode	Meaning
ldac *opd*	Load the address of the constant *opd*
ldal *opd*	Load the address of the local variable whose offset from the frame pointer is *opd*
ldar *opd*	Load the address of register *opd*
ldatls *opd*	Load the address of the thread local storage object at offset *opd* in the thread local space
ldc *opd*	Load the constant *opd*
ldgtls *opd*	Load the address of the general thread local storage object whose key is *opd*
ldl *opd*	Load the value of the local variable whose offset from the frame pointer is *opd*
ldm *opd*	Load the value stored in memory at address *opd*
ldr *opd*	Load the contents of register *opd*

The following special opcode is most often used in addressing expressions that are appended to existing addressing expressions:

TABLE 4: **Special Opcode**

Opcode	Meaning
remove_-indirection	Removes an indirection operation from the tail of the previous addressing expression; this is useful when you for backing up from data to a dope vector.

Creating Vector Transformations

2

This chapter is a detailed examination of how to create an STL vector transformation. It also discusses the TTF convenience routines that help create the vector transformation. After reading this chapter, you should understand how you go about creating a transformation and the issues involved when you create your own. As you will see, the problems that exist when you create a transformation for your own data types are unique and there are no easy solutions.

Note from the Author: You are encouraged to read this chapter using the PDF or HTML versions. This chapter makes extensive use of links so that you can click on Tcl procedure names and be taken to the procedure's description. This should make it easier to understand this chapter's contents.

Non-vector Transformations

While the subject of this chapter is vector transformations, you can also create list and struct transformations. (While you can create your own map transformations, it is not recommended.) The information in this chapter is a starting point. After you understand this information, you can go to our **lib** subdirectory and view how Etnus implemented these transformations for your system. From within the CLI, you can obtain the location of this library's directory by typing:

```
dset TOTALVIEW_TCLLIB_PATH
```

The Vector Transformation

This vector transformation has the following procedures:

- "vector_validate"
- "vector_type" on page 18
- "vector_lower_bounds" on page 19
- "vector_extent" on page 19
- "vector_addressing" on page 21

vector_validate

This procedure validates the layout of the internal representation of a GCC vector. This representation is:

```
vector                     class vector
  _Vector_base<int,allo.. class _Vector_base<..
                   (Protected base class)
     _Vector_alloc_base..  class _Vector_alloc..
                   (Public base class)
        _M_start          int*
        _M_finish         int*
        _M_end_of_storage int*
```

The validation routine checks the layout of data type that matched the regular expression to make sure that it is processing what it expected to be processing. Along the way, this routine obtains the soid (symbol object ID) of the target type index for the type of Vector and also the soids of the **_M_start** and **_M_finish** members. At a later time, another callback will use these indices to compute the vector's bounds.

The information needed at a later time is stored in a global array. Here are the elements that this routine stores:

- **vector_type_id**: The soid of the target type for the vector.
- **_Vector_base_id**: The soid for the **_Vector_base** class
- **_Vector_alloc_base_id**: The soid for the **_Vector_alloc_base** class.
- **_Vector_alloc_base_M_start_id**: The soid for the **_M_start** data type.
- **_Vector_alloc_base_M_start_location**: The "formula" to get to the start of the vector. This computes, starting from the top of the internal Vector structure the offset to **_M_start**.
- **_Vector_alloc_base_M_finish_id**: The soid for the **_M_finish** data type.
- **_Vector_alloc_base_M_finish_loc**: The "formula" to get to the end of the vector. This computes, starting from the top of

This validation routine is rather lengthy. However, all it does is go from class to class and member to member within the vector's structure. It also saves layout information while it does this.

vector

_Vector_base

```
proc vector_validate {symbol_id} {
    # The incoming symbol_id (soid) has already matched a regular
    # expression that indicates that this symbol looks like a GCC
    # vector. It has the form  vector<int,allocator<int> >. So, do
    # some simple checking to make sure it really is a GCC vector.
    # Make sure that this file was compiled by the GNU compiler.
    if {![::TV::TTF::ttf_check_symbol_compiler \
              $symbol_id "gnu_v2"] &&
            ![::TV::TTF::ttf_check_symbol_compiler \
              $symbol_id "gnu_v3"] } {
        return false
    }
        # Make sure incoming symbol is of kind "aggregate_type".
    if {![::TV::TTF::ttf_is_symbol_of_kind \
              $symbol_id "aggregate_type"]} {
        return false
    }

        # Make sure that the external name for this symbol is some-
        # thing like vector<...>. In other words, this revalidates
        # the regular expression matching that caused this
        # procedure to be activated. This isn't strictly necessary.
        #
        # For example, this could return:
        #        class vector<int,allocator<int> >
    if {![regexp {^(class|struct) (std::)?vector *<.*>$} \
            [::TV::TTF::ttf_get_symbol_external_name \
              $symbol_id] match]} {
        return false
    }
    # The next set of operations begins analyzing the vector's
    # structure. The first step is to locate information about the
    # _Vector_base class that vector extends. It begins by
    # obtaining the symbol ID for the vector's base class. For
    # example the value returned might be something like
    # "1|26".
    #
    # You will need to spend some time understanding how
    # ttf_get_base_class_id  works before you can write your own
    # transformations.
    set _Vector_base_id [::TV::TTF::ttf_get_base_class_id \
              $symbol_id]
    if { $_Vector_base_id == "" } {
        return false
    }

        # Store the ID of _Vector_base.
    set analysis_info("_Vector_base_id") $_Vector_base_id

        # Get the location offset of the base class from this class so
        # we can use it when we need to access the member. For a
        # vector, ttf_get_base_class_location returns {}. In turn,
        # ttf_check_location returns "addc 0". This will be the first,
        # addressing expression. In other words, _Vector_alloc_base
        # is not using any storage.
    set _Vector_base_location ""
    append _Vector_base_location \
        "{" \
        [::TV::TTF::ttf_check_location
```

```
          [::TV::TTF::ttf_get_base_class_location \
                $symbol_id]] \
    "} "
```

_Vector_alloc_-base

```
# Move down to the _Vector_alloc_base class that
# _Vector_base extends.
#
# Notice that the code for analyzing this class is identical to that
# which was used for the previous class. And, the results are the
# same: it creates an "{addc 0}" addressing expression.

    # Get the symbol ID for the base class to _Vector_base.
set _Vector_alloc_base_id \
        [::TV::TTF::ttf_get_base_class_id $_Vector_base_id]
if { $_Vector_alloc_base_id == "" } {
    return false
}

    # Store off the ID of the _Vector_base.
set analysis_info("_Vector_alloc_base_id") \
          $_Vector_alloc_base_id

    # Get the location offset of the base class from this class.
    # This is used when accessing members.
set _Vector_alloc_base_location ""
append _Vector_alloc_base_location \
    "{" \
    [::TV::TTF::ttf_check_location \
        [::TV::TTF::ttf_get_base_class_location \
              $_Vector_base_id]] \
    "} "
```

_Vector_alloc_-base member and _M_start analysis

```
# Finally, the vector_validate procedure is ready to look at the
# individual members of _Vector_alloc_base, which is where the
# vector's data is. There are three members: _M_start,
# _M_finish, and _M_end_of_storage. Only the first two are
# important as they let us compute the vector's bounds.

# Get the _M_start data member. The returned value will be a
# symbol such as "1|30".
#
# When writing your own transformations, you'll have to
# understand how the TTF routines used  here works.
set _Vector_alloc_base_M_start_id \
    [::TV::TTF::ttf_get_single_symbol_id_from_scope \
          $_Vector_alloc_base_id "member" "_M_start"]
if { $_Vector_alloc_base_M_start_id == "" } {
    return false
}

# Get the location of _M_start. This address is relative to the
# previous two addresses. In other words, what you need to do is
# append the address of _M_start to the previous two addresses.
# The result will be {addc 0}{addc 0}{something}. In this case,
# we obtain yet another {addc 0}.  This final addc is returned by
# the ttf_check_location routine.
#
# This is an instance of us being very, very cautious. Since you
# know that this is 0, you could just ignore it.
```

```
set _Vector_alloc_base_M_start_location ""
append _Vector_alloc_base_M_start_location \
    $_Vector_base_location \
    $_Vector_alloc_base_location \
    "{" \
    [::TV::TTF::ttf_check_location \
        [TV::symbol get \
            $_Vector_alloc_base_M_start_id location]] \
    "} "

    # Store off information about the _M_start member.
set analysis_info("_Vector_alloc_base_M_start_id") \
    $_Vector_alloc_base_M_start_id
set analysis_info("_Vector_alloc_base_M_start_location") \
    $_Vector_alloc_base_M_start_location

    # Determine the type of the vector by analyzing the type of
    # the _M_start member. This is actually a pointer to the
    # actual data type of the vector. This means that we will need
    # to resolve this to the actual type of the list. The returned
    # value looks something like: <2,0,409>.
set _Vector_alloc_base_M_start_type_index \
    [TV::symbol get $_Vector_alloc_base_M_start_id \
          type_index]

    # Get the containing image ID for the symbol.
set image_id [::TV::TTF::ttf_get_containing_image_id \
             $symbol_id]

    # Get the symbol ID for _M_start.
set _Vector_alloc_base_M_start_type_id \
    [TV::scope lookup $image_id in_scope \
        $_Vector_alloc_base_M_start_type_index]

    # Make sure what TotalView returned is a "pointer_type".
if {[TV::symbol get $_Vector_alloc_base_M_start_type_id \
                    kind] \
        != "pointer_type"} {
    return false
}

    # Get the target type index for the _M_start symbol and
    # then get the ID for it. We'll store this ID off for later use.
set target_type_index \
[TV::symbol get $_Vector_alloc_base_M_start_type_id \
          target_type_index]
set target_type_id \
  [TV::scope lookup $image_id in_scope \
          $target_type_index]

    # Make sure the target type is fully resolved.
set target_type_id \
          [TV::type resolve_final $target_type_id]

    # Store off information about the target type of _M_start
    # member.
set analysis_info("vector_type_id") $target_type_id
```

_M_finish

```
# Get the _M_finish data member. This address is relative to the
# previous two class addresses. It is not relative to the _M_start,
# member. This address will be appended to the two addresses
# for the classes, both of which were {addc 0}. The result is
# {addc 0}{addc 0}{something}. In this case, this is {addc 4}.
# This final addc is returned by the ttf_check_location routine.

# Notice that the routines in this section are identical to those
# used in the previous section. And, like for the _M_start
# routine, the {addc 0} expressions are there because we're
# being careful. If you know that something will always be zero,
# you need not include it.
set _Vector_alloc_base_M_finish_id \
    [::TV::TTF::ttf_get_single_symbol_id_from_scope \
        $_Vector_alloc_base_id "member" "_M_finish"]
if { $_Vector_alloc_base_M_finish_id == "" } {
    return false
}

    # Get the location of _M_finish.
set _Vector_alloc_base_M_finish_loc ""
append _Vector_alloc_base_M_finish_loc \
    $_Vector_base_location \
    $_Vector_alloc_base_location \
    "{" \
    [::TV::TTF::ttf_check_location \
        [TV::symbol get \
            $_Vector_alloc_base_M_finish_id location]] \
    "} "

    # Store off information about the _M_finish member.
set analysis_info("_Vector_alloc_base_M_finish_id") \
    $_Vector_alloc_base_M_finish_id
set analysis_info("_Vector_alloc_base_M_finish_loc") \
    $_Vector_alloc_base_M_finish_loc
```

Final steps

```
# Save the extracted information from the types so it can be
# accessed later. As there can be more than one variable
# associated with a transformation, it will be associated with the
# incoming symbol ID. As TotalView passes this ID to other
# callbacks, you can retrieve this data by using this ID.
variable _vector_type_info
set _vector_type_info($symbol_id) \
        [array get analysis_info]

    # Made it through all the checks. The GCC Vector is what we
    # expected!
return true
}
```

vector_type

Return the type ID for the target type. This is the "type" of the vector such as **int**. All this routine is doing is returning the value created by the **vector_valid** routine.

```
proc vector_type {symbol_id} {
    variable _vector_type_info

    array set analysis_info $_vector_type_info($symbol_id)

    return $analysis_info("vector_type_id")
}
```

vector_lower_bounds

Create the addressing expression that determines the offset for the lower bounds for the given type ID. For C/C++, the vector's lower bound is always 0, so all that needs to be done is **dup** the accumulator and subtract it from itself to yield 0.

Because TotalView will send a **symbol_id** to the routine, it is used as the procedure's parameter even though it isn't used.

```
proc vector_lower_bounds {symbol_id} {
    return [list dup minus value]
}
```

vector_extent

Create the expression that determines the offset for the upper bounds for the given type ID. This is the most difficult of the routines. This code presentation is immediately followed by a table that describes just the addressing expression being created and what it does.

```
proc vector_extent {symbol_id} {
    variable _vector_type_info

    array set analysis_info $_vector_type_info($symbol_id)
    set lower_bound_location \
        $analysis_info("_Vector_alloc_base_M_start_location")
    set upper_bound_location \
        $analysis_info("_Vector_alloc_base_M_finish_loc")
    set target_type_id $analysis_info("vector_type_id")

        # For GCC, the offset is the difference between the
        # addresses of _M_start and _M_finish divided by the size of
        # the vector's type. That is:
        #
        # (_M_finish - _M_start)/size
        #
        # Dup the TOS. This preserves the original ACC and the one
        # we will operate upon. This will be before the upper bound.
    set location {}
    lappend location "dup"

        # This adds in addressing expressions to locate to _M_finish;
        # for example, {addc 0}{addc 0}{addc 4}. Loosely speaking,
        # only the {addc 4} is necessary.
    set location [concat $location $upper_bound_location]

        # Change ACC into an actual address.
    lappend location "indirect"

        # Swap position of the address and original ACC.
    lappend location "swap"

        # This adds in addressing expressions to locate to _M_start;
        # for example, {addc 0}{addc 0}{addc 0}. Loosely speaking,
        # only one {addc 0} is needed.
    set location [concat $location $upper_bound_location]
    set location [concat $location $lower_bound_location]
```

```
        # Change ACC into an actual address. Now at this point we
        # should have the actual address of the upper bound and
        # lower bound on the stack. Taking the difference of these
        # will yield the extent.
    lappend location "indirect"

        # Final value is the extent times the size of the target type.
    lappend location "minus"

        # Divide this value by the target type size. Push the size of
        # the target type onto stack.
    set target_type_length \
            [TV::symbol get $target_type_id length]
    lappend location "ldc $target_type_length"

        # Divide to determine actual extent.
    lappend location "div"

        # Finally specify that this is actually the value to use and not
        # use it as an address.
    lappend location "value"
    return $location
}
```

This procedures is doing something really simple. Unfortunately, the translation of what is something that is simple into terms that TotalView can understand gets a little complicated. This routine is just subtracting the first address where data is stored from the second address where data is stored, then dividing this number by the word size. That is:

(address1 - address2)/word_size

The result is the number of instances in the vector.

Here, using the components created by the callbacks, is the addressing expression that performs this operation:

```
dup {addc 0} {addc 0} {addc 4} indirect swap
{addc 0} {addc 0} {addc 0} indirect
minus {ldc 4} div value
```

Just to make it a little simpler, lets assume that it is:

```
dup {addc 4} indirect swap indirect minus {ldc 4} div value
```

In other words, the **{addc 0}** statements that don't change the address have been eliminated.

TABLE 1: Figuring out the Vector Extent

	Op	Stack	Location
1.	—	value	stack[0] (ACC)
2.	`dup`	value	stack[0]
		value	stack[1] (ACC)
			The value is duplicated.
3.	`{addc 4}`	value	stack[0]
		value+4	stack[1] (ACC)
			Note: **addc** is defined as follows:
			`ACC = ACC + constant`
			4 is added to the accumulator

	Op	Stack	Location
4.	`indirect`	value	stack[0]
		addrE	stack[1] (ACC)
			Note: indirect is defined as follows:
			`memory[ACC]`
			The accumulator now points to the value in an address.
5.	`swap`	addrE	stack[0]
		value	stack[1] (**ACC**)
			Note: **swap** changes the positions of the last two entries on the stack and the ACC stays as the last entry on the stack.
6.	`indirect`	addrE	stack[0]
		addrS	stack[1] (ACC)
7.	`minus`	(addrE-addrS)	stack[0] (ACC)
			Note: **minus** is defined as follows:
			`ACC = stack[depth-1] - ACC`
			So in this case, the **minus** operation is: `ACC = stack[0] - ACC`
			That is:
			`ACC = addrE - addrS`
			That is, we now have a value that is the difference between these two addresses.
8.	`{ldc 4}`	(addrE-addrS)	stack[0]
		4	stack[1] (ACC)
			Set the accumulator to 4.
9.	`div`	(addrE-addrS)/4	stack[0] (TOS)
			Note: **div** is defined as follows:
			`ACC = stack[depth-1]/ACC`
			So in this case, the **div** operation is:
			`ACC = stack[0]/4`
			That is:
			`ACC = (addrE-addrS)/4`
10.	`value`	(addrE-addrS)/4	stack[0] (TOS)
			The value at the TOS is treated as a number.

vector_-addressing

Returns the addressing expression for the vector. This provides a “formula” to access **_M_start**, which is the first element of the vector.

```
proc vector_addressing {symbol_id} {
    variable _vector_type_info
    array set analysis_info $_vector_type_info($symbol_id)
    set lower_bound_location $analysis_info\
        (“_Vector_alloc_base_M_start_location”)
```

```
        # For GCC it is simply address of _M_start.
    set location {}

        # This adds in addressing expressions to set to _M_start; for
        # example, {addc 0}{addc 0}{addc 0}.
    set location [concat $location $lower_bound_location]

        # Change TOS into an actual address.
    lappend location "indirect"
    return $location
}
```

Final Steps

Now that everything is defined, create and install the STL vector transformation.

```
set type_transformation_id \
        [TV::type_transformation create Array]

TV::type_transformation set $type_transformation_id \
    name          {^(class|struct) (std::)?vector *<.*>$} \
    language                        C++ \
    type_transformation_description "GNU Vector"\
    validate_callback               vector_validate \
    lower_bounds_callback           vector_lower_bounds \
    upper_bounds_callback           vector_extent \
    addressing_callback             vector_addressing \
    type_callback                   vector_type
```

Convenience routines

The convenience routines are Tcl CLI procedures that take much of the drudgery out creating transformations as they extract symbol and scope information for you.

The routines discussed in this section are:

- "dump_type_transformation" on page 23
- "ttf_check_location" on page 23
- "ttf_check_symbol_compiler" on page 23
- "ttf_debug_puts" on page 23
- "ttf_extract_offset" on page 24
- "ttf_get_base_class_id" on page 24
- "ttf_get_base_class_location" on page 25
- "ttf_get_containing_image_id" on page 25
- "ttf_get_single_symbol_from_scope" on page 26
- "ttf_get_single_symbol_id_from_scope" on page 26
- "ttf_get_symbol_external_name" on page 27
- "ttf_is_symbol_of_kind" on page 27
- "ttf_read_store" on page 27
- "ttf_resolve_final_type_index" on page 27
- "ttf_resolve_target_type" on page 28

dump_type_transformation

Dump out all of a type transformation's properties and values. It is a good idea to call this routine right after you instantiate a transformation.

```
proc dump_type_transformation {id} {
    foreach prop [TV::type_transformation properties] {
        ttf_debug_puts [format "%-25s %s" $prop \
            [TV::type_transformation get $id $prop]]
    }
}
```

ttf_check_location

Given a location of the form {**addc n**}, strip off the braces { } and return **addc n**. If an empty location is passed in, indicating 0, it returns **addc 0**.

```
proc ttf_check_location {location} {
    if {[string length $location] == 0} {
        return "addc 0"
    } else {
        regexp "(\[a-z\]+\[ \]*\[0-9\]*)" $location match
        return [string trim $match]
    }
}
```

ttf_check_symbol_compiler

Check to insure that the source file was compiled using the compiler for which a transformation is associated.

```
proc ttf_check_symbol_compiler {symbol_id compiler} {
        # Walk up the scopes until the containing file is found.
    set kind [TV::symbol get $symbol_id kind]
    set file_id $symbol_id
    while {$kind != "file"} {
        set file_id [TV::symbol get $file_id scope_owner]
        set kind [TV::symbol get $file_id kind]
    }

        # Get the compiler used on the file.
    set compiler_kind [TV::symbol get $file_id compiler_kind]

      # See if the compiler kind matches the incoming one.
    if { $compiler_kind != $compiler } {
        return 0
    }
    return 1
}
```

ttf_debug_puts

When the **_ttf_debug** global variable is set to true, display TTF-related debugging output.

```
proc ttf_debug_puts {{string ""}} {
    variable _ttf_debug

    if {$_ttf_debug} {
        puts $string
    }
}
```

ttf_extract_offset

Given an addressing expression that will only contain "**addc** *n*", return *n*.

```
proc ttf_extract_offset {addressing_expr} {
    if {[llength $addressing_expr] != 1} {
        return 0
    }

        # Unwind the list.
    set addressing_expr [lindex $addressing_expr 0]
    if {[lindex $addressing_expr 0] != "addc"} {
        return 0
    } else {
        return [lindex $addressing_expr 1]
    }
}
```

ttf_get_base_-class_id

Find the actual base class of a symbol. This assumes that only a single base class exists for the symbol.

This procedure obtains the base class member of the given **symbol_id**. This is not, however, the actual base class. To get it, we need to get the **type_index** of this base class member and then look up the corresponding symbol for it.

```
proc ttf_get_base_class_id {symbol_id} {
        # Get the base class member of the symbol_id.
        # ttf_get_single_symbol_from_scope returns a list of
        # information.
    set base_class_symbol \
        [ttf_get_single_symbol_from_scope \
                $symbol_id "member" "!base_class"]
    if { $base_class_symbol == "" } {
        return ""
    }

        # From this list, extract the value of the "id" sublist. For
        # example, a value such as "1|25" might be returned.
    if {![regexp {(id )([0-9]+\|[0-9]+)} \
          $base_class_symbol match tag base_class_symbol_id]} {
        return ""
    }

        # Get the type_index of the symbol. This will be a triple that
        # looks something like "<2,0,49>". TotalView uses this triple
        # to locate information that it stores about your program's
        # symbols.
    set type_index \
            [TV::symbol get $base_class_symbol_id type_index]

        # Get the containing image ID for the symbol. (An image can
        # be thought of as the set of processes being run that make
        # up your program.) The returned value will look something
        # like "1|24".
    set image_id \
            [ttf_get_containing_image_id $symbol_id]

        # When TotalView reads the image, it created an entry in its
        # internal symbol table for all of your program's data types.
```

```
        # Now that it has located the image_id, it can now locate the
        # internal ID of the data type.
    set base_class_symbol_ids \
        [capture TV::scope lookup $image_id in_scope \
                $type_index]
    if {[llength $base_class_symbol_ids] != 1} {
                # Did not find the correct number!
        return ""
    }

        # Get the actual base class ID.
    set base_class_id [lindex $base_class_symbol_ids 0]

        # Make sure that TotalView has the final type. You need to
        # do this because TotalView may defer reading in all
        # information about the symbol until it actually needs to
        # use the information.
    set base_class_id [TV::type resolve_final $base_class_id]

        # Return the ID.
    return $base_class_id
}
```

ttf_get_base_class_location

Look up the location offset of the base class associated with a symbol. This assumes only a single base class for the given symbol. That is, this is undefined if you are using multiple inheritance for a data type.

```
proc ttf_get_base_class_location {symbol_id} {
        # Get the base class member of the given symbol_id. This
        # routine returns a list of the symbol's attributes.
    set base_class_symbol \
            [ttf_get_single_symbol_from_scope \
                $symbol_id "member" "!base_class"]
    if { $base_class_symbol == "" } {
        return ""
    }

        # From this list, extract the ID's value.
    if {![regexp {(id )([0-9]+\|[0-9]+)} \
        $base_class_symbol match tag base_class_symbol_id]} {
        return ""
    }

        # Return the location property.
    return [TV::symbol get $base_class_symbol_id location]
}
```

ttf_get_containing_image_id

Given a valid symbol ID, recursively walk backwards up the scope until it locates the containing image for the symbol.

```
proc ttf_get_containing_image_id {symbol_id} {
        # Check the kind and see if this is an image. If it is, we're
        # done.
    set base_kind [TV::symbol get $symbol_id kind]
    if {$base_kind == "image"} {
            # Get the soid of the image.
        set image_id [TV::symbol get $symbol_id id]
        return $image_id
```

```
        }

        # Recurse using the scope_owner.
        set scope_owner [TV::symbol get $symbol_id scope_owner]
        return [ttf_get_containing_image_id $scope_owner]
}
```

ttf_get_single_symbol_from_scope

Given a symbol that is a scope, locate a single symbol from within its scope of symbols. This procedure uses the **kind** and **base_name** properties of the symbol to match the desired symbol.

```
proc ttf_get_single_symbol_from_scope \
            {symbol_id kind base_name} {

        # Get all the symbols in the scope. For this vector, there are
        # three sets of information: one for _M_start, _M_finish, and
        # _M_end_of_storage.
    set symbols [split [string trim \
            [capture TV::scope dump $symbol_id] "\n"] "\n"]
    foreach symbol $symbols {
            # Get the ID (soid) of the symbol.
        if {![regexp {(id )([0-9]+\|[0-9]+)} \
                $symbol match tag soid]} {
            continue
        }

            # Get the kind of the symbol. For example, look for
            # _M_start.
        set symbol_kind [TV::symbol get $soid kind]
        if {$symbol_kind != $kind} {
            continue
        }

            # Get the base_name of the symbol.
        set symbol_base_name [TV::symbol get $soid base_name]
        if {$symbol_base_name != $base_name} {
            continue
        }

            # The kind and base_name match. This is the symbol
            # being looked for.
        return $symbol
    }

        # We've fallen through the loop without finding anything.
    return ""
}
```

ttf_get_single_symbol_id_from_scope

Look up the symbol within the scope based upon the **kind** and **base_name** and returns the id of the found symbol.

```
proc ttf_get_single_symbol_id_from_scope \
         {symbol_id kind base_name} {

        # The next statement returns a list that looks something like:
        #    {kind member} {id 1|78} ... {type_index <2,0,409>}
        # All we're going to do is extract the "id" component of the
        # list.
```

```
    set symbol [ttf_get_single_symbol_from_scope \
                $symbol_id $kind $base_name]

        # What we have is the raw symbol. Obtain the ID from it.
    if {![regexp {(id )([0-9]+\|[0-9]+)} \
            $symbol match tag symbol_id]} {
        return ""
    }
    return $symbol_id
}
```

ttf_get_symbol_-external_name

Return the **external_name** of a symbol. For example, here is what was returned when this routine was manually tested:

class vector<char *,allocator<char *> >

```
proc ttf_get_symbol_external_name {symbol_id} {
    return [TV::symbol get $symbol_id external_name]
}
```

ttf_is_symbol_-of_kind

Check to see if the symbol is of the specified **kind**.

```
proc ttf_is_symbol_of_kind {symbol_id kind} {
    set symbol_kind [TV::symbol get $symbol_id kind]
    if {$symbol_kind != $kind} {
        return 0
    }

    return 1
}
```

ttf_read_store

Read a value from an absolute address.

```
proc ttf_read_store {address {type long}} {
    set res [capture dprint "*($type *)$address"]
        # Strip out just the value.
    regexp {^.*= ([^ ]*)} $res null res

    return $res
}
```

ttf_resolve_-final_type_index

After resolving a **target_type_index**, return its **type_index**. That is, some symbols only serve to hold a reference to another symbol. For example, a **typedef** is a reference to the aliased type. Similarly, a **const**-qualified type is a reference to the non-**const**s qualified type. These reference types are called undiscovered symbols. This operation, when performed on an undiscovered symbol, returns the symbol the type refers to. This allows it to return that symbol's **type_index**.

```
proc ttf_read_store {address {type long}} {
        # Gets the ID of the target type index.
    set ids [capture TV::scope lookup $image_id in_scope \
            $target_type_index]
    if {[llength $ids] != 1} {
        # Did not find the correct number.
        return ""
    }
```

```
        # Get the actual base class ID.
    set id [lindex $ids 0]

        # Resolve to the final type of the ID.
    set id [TV::type resolve_final $id]

        # Return the target type index of the final ID.
    return [TV::symbol get $id type_index]
}
```

ttf_resolve_-target_type

Return the ID of the target type, resolved to a non-pointer type.

```
proc ttf_resolve_target_type {type_index image_id} {
        # Look up the ID of the type index.
    set type_id \
            [TV::scope lookup $image_id in_scope $type_index]

        # Resolve the type back to base type.
    set base_type_id [TV::type resolve_final $type_id]

        # Make sure that a kind of "pointer_type" was returned.
    if {[TV::symbol get $base_type_id kind] != "pointer_type"}
    {
       return $base_type_id
    }

        # Determine what the actual type is by making sure all sym-
        # bols are read.
    TV::symbol read_delayed $base_type_id

        # Get the target type index for the base type.
    set target_type_index \
            [TV::symbol get $base_type_id target_type_index]

        # Look up the ID of the type.
    set target_type_id \
      [TV::scope lookup $image_id in_scope $target_type_index]

        # Test before returning to prevent opaque_type from
        # returning. This is a TotalView bug.
    if {[TV::symbol get $target_type_id kind] == \
            "opaque_type"} {
       return false
    }

        # See if the type is undiscovered. It so, resolve it.
    if {[TV::symbol get $target_type_id kind] == \
                "ds_undiscovered_type" } {
        set target_type_id \
                [TV::type resolve_final $target_type_id]
    }
    return $target_type_id
}
```

TTF CLI Commands 3

When you create a type transformation, you will make extensive use of the **TV::scope** and **TV::symbol** commands. In addition, you may need to use the **TV::type** command.

After you have created your callbacks, you will use the **TV::type_transformation** command to install it.

Here is where you will find these commands:

The information presented on **TV::type** duplicates information found in the TOTALVIEW REFERENCE GUIDE. In contrast, the other three are not described in that book.

scope

Returns information about a symbol's scope

Format: **TV::scope** *action* [*object-id*] [*other-args*]

Arguments: *action* The action to perform, as follows:

cast Attempts to find or create the type named by the *other-args* argument in the given scope.

commands Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

dump Dump all properties of all symbols in the scope and in the enclosed scope.

get Returns properties of the symbols whose soids are specified. Specify the kinds of properties using the *other-args* argument.

If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.

lookup Look up a symbol by name. Specify the kind of lookup using the *other-args* argument. The values you can enter are:

by_language_rules: Use the language rules of the language of the scope to find a single name.

by_path: Look up a symbol using a pathname.

by_type_index: Look up a symbol using a type index.

in_scope: Look up a name in the given scope and in all enclosing scopes, and in the global scope.

lookup_keys Displays the kinds of lookup operations that you can perform.

properties Displays the properties that the CLI can access. Do not use additional arguments with this option. The arguments displayed are those that are displayed for the scope of all types. Additional properties also exist but are not shown.(Only the ones used by all are visible.) For more information, see **TV::symbol**.

walk Walk the scope, calling Tcl commands at particular points in the walk. The commands are named using the following options:

–pre_scope *tcl_cmd*: Names the commands called before walking a scope.

–pre_sym *tcl_cmd*: Names the commands called before walking a symbol.

	–post_scope *tcl_cmd*: Names the commands called after walking a scope.
	–post_symbol *tcl_cmd*: Names the commands called after walking a symbol.
	tcl_cmd: Names the commands called for each symbol.
object-id	The ID of a scope.
other-args	Arguments required by the **get** subcommand.

Description: The **TV::scope** command lets you examine and set a scope's properties and states.

You'll find many examples of this command being used in Chapter 2, "Creating Vector Transformations," on page 13.

symbol

Returns or sets internal TotalView symbol information

Format: **TV::symbol** *action* [*object-id*] [*other-args*]

Arguments: *action* The action to perform, as follows:

commands Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

dump Dumps all properties of the symbol whose soid (symbol object ID) is named. Do not use additional arguments with this command.

get Returns properties of the symbols whose soids are specified here. The *other-args* argument names the properties to be returned.

properties Displays the properties that the CLI can access. Do not use additional arguments with this option. These properties are discussed later in this section.

read_delayed Only global symbols are initially read; other symbols are only partially read. This command forces complete symbol processing for the compilation units that contain the named symbols.

resolve_final Performs a sequence of **resolve_next** operations until the symbol is no longer undiscovered. If you apply this operation to a symbol that is not undiscovered, it returns the symbol itself.

resolve_next Some symbols only serve to hold a reference to another symbol. For example, a **typedef** is a reference to the aliased type, or a **const**-qualified type is a reference to the non-**const**s qualified type. These reference types are called *undiscovered symbols.* This operation, when performed on an undiscovered symbol, returns the symbol the type refers to. When this is performed on a symbol, it returns the symbol itself.

rebind Changes one or more structural properties of a symbol. These operations can crash TotalView or cause TotalView to produce inconsistent results. The properties that you can change are:

address: the new address:

base_name: the new base name. The symbol must be a base name.

line_number: the new line number. The symbol must be a line number symbol.

loader_name: the new loader name and a file name.

scope: the soid of a new scope owner.

type_index: the new type index, in the form **<n, m, p>**. The symbol must be a type.

object-id The ID of a symbol.

other-args Arguments required by the **get** subcommand.

Description: The **TV::symbol** command lets you examine and set the symbol properties and states.

Symbol Properties The following table lists the properties associated with the symbols information that TotalView stores. Not all of this information will be useful when creating transformations. However, it is possible to come across some of these properties and this information will help you decide if you need to use it in your transformation. In general, the properties used in the transformation files that Etnus provided will be the ones that you will use.

TABLE 1: **Symbol Properties**

Symbol Kind	Has base_name	Has type_index	Property		
aggregate_type	✔	✔	aggregate_kind artificial external_name	full_pathname id kind	length logical_scope_owner scope_owner
array_type	✔	✔	artificial data_addressing element_addressing external_name full_pathname id	index_type_index kind logical_scope_owner lower_bound scope_owner stride_bound	submembers target_type_index upper_bound validator
block	✔		address_class artificial full_pathname	id kind length	location logical_scope_owner scope_owner
char_type	✔	✔	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
code_type	✔	✔	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
common	✔		address_class artificial full_pathname	id kind location	logical_scope_owner scope_owner
ds_undis-covered_type	✔	✔	artificial full_pathname id	kind logical_scope_owner scope_owner	target_type_index
enum_type	✔	✔	artificial enumerators external_name	full_pathname id kind	logical_scope_owner scope_owner value_size

TABLE 1: **Symbol Properties**

Symbol Kind	Has base_name	Has type_index	Property		
error_type	✔	✔	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
file	✔		artificial compiler_kind delayed_symbol demangler	full_pathname id kind language	logical_scope_owner scope_owner
float_type	✔	✔	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
function_type	✔	✔	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
image	✔		artificial full_pathname	id	kind
int_type	✔	✔	artificial external_name full_pathname	id kind length	logical_scope_owner scope_owner
label	✔		address_class artificial full_pathname	id kind location	logical_scope_owner scope_owner
linenumber			address_class artificial full_pathname	id kind location	logical_scope_owner scope_owner
loader_symbol			address_class artificial full_pathname	id kind length	location logical_scope_owner scope_owner
member	✔		address_class artificial full_pathname id	inheritance kind location logical_scope_owner	ordinal scope_owner type_index
module	✔		artificial full_pathname	id kind	logical_scope_owner scope_owner
named_constant	✔		artificial full_pathname id	kind length logical_scope_owner	scope_owner type_index value
namespace	✔		artificial full_pathname	id kind	logical_scope_owner scope_owner
opaque_type	✔	✔	artificial external_name full_pathname	id kind logical_scope_owner	scope_owner
pathname_-reference_-symbol	✔		artificial id full_pathname	kind lookup_scope logical_scope_owner	resolved_symbol_-pathname scope_owner
pointer_type		✔	artificial external_name full_pathname id	kind length logical_scope_owner scope_owner	target_type_index validator

TABLE 1: **Symbol Properties**

Symbol Kind	Has base_name	Has type_index	Property		
qualified_type	✔	✔	artificial	id	qualification
			external_name	kind	scope_owner
			full_pathname	logical_scope_owner	target_type_index
soid_reference_-symbol	✔		artificial	kind	scope_owner
			full_pathname	logical_scope_owner	
			id	resolved_symbol_id	
stringchar_type	✔	✔	artificial	id	scope_owner
			external_name	kind	
			full_pathname	logical_scope_owner	
subroutine	✔		address_class	kind	return_type_index
			artificial	length	scope_owner
			full_pathname	location	static_chain
			id	logical_scope_owner	static_chain_height
typedef	✔	✔	artificial	id	logical_scope_owner
			external_name	kind	scope_owner
			full_pathname	length	target_type_index
variable	✔		address_class	is_argument	ordinal
			artificial	kind	scope_owner
			full_pathname	location	type_index
			id	logical_scope_owner	
void_type	✔	✔	artificial	id	logical_scope_owner
			external_name	kind	scope_owner
			full_pathname	length	

The figure on the following page shows how these symbols are related. Here are definitions of the properties associated with these symbols.

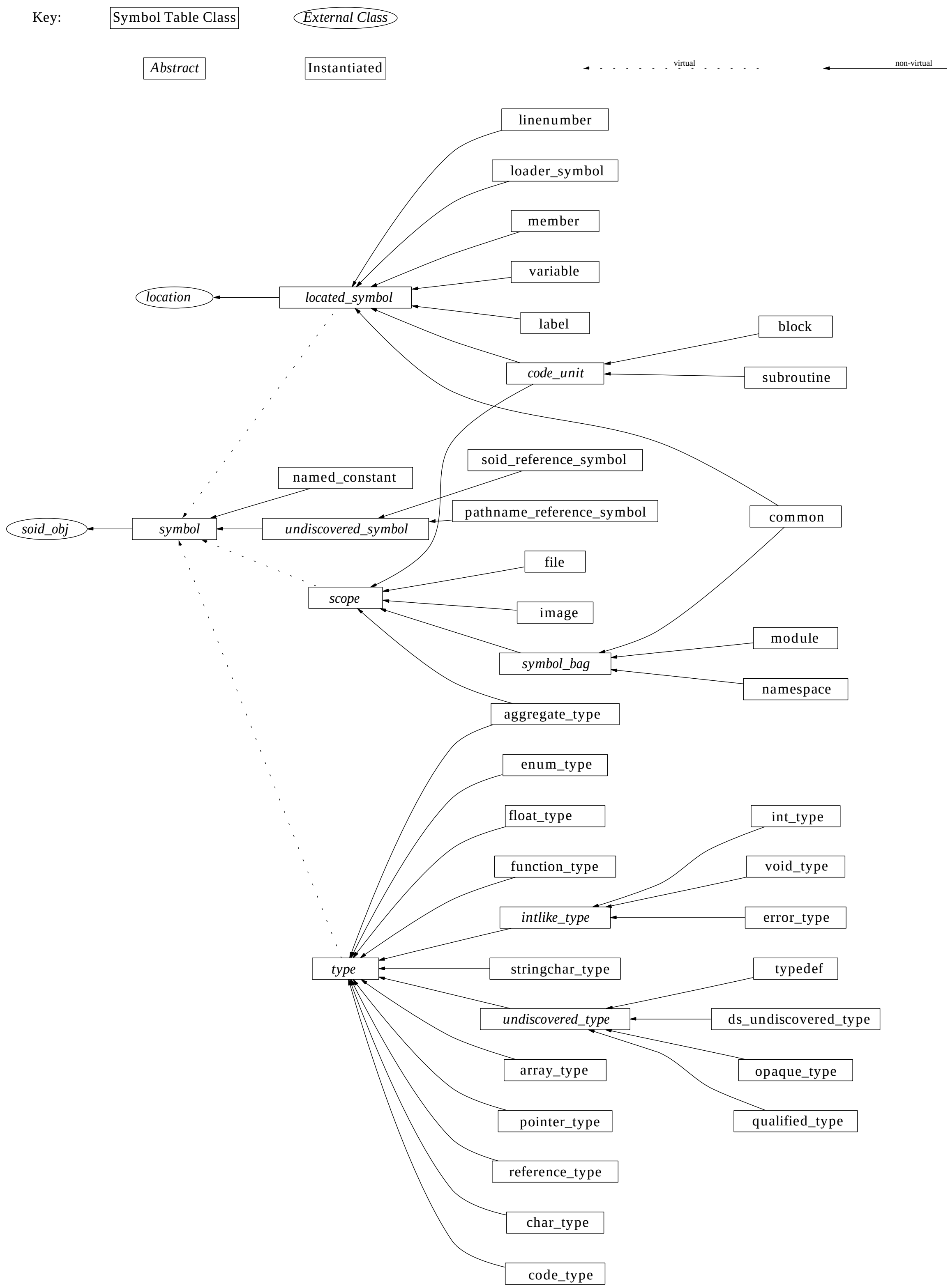
address_class contains the location for a variety of objects such as a **func**, **global_var**, and a **tls_global**.

aggregate_kind One of the following: **struct**, **class**, or **union**.

artificial A Boolean (0 or 1) value where true indicates that the compiler generated the symbol.

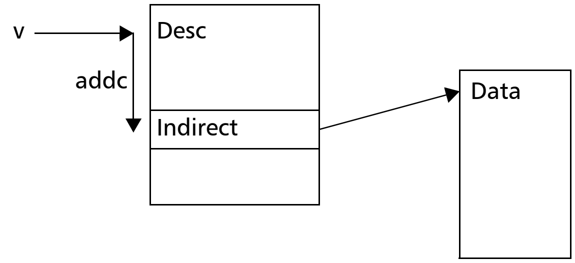
compiler_kind The compiler or family of compiler used to create the file. For example, **gnu**, **xlc**, **intel**, and so on.

data_addressing Contains additional operands to get from the base of an object to its data. For example, a Fortran by-desc array contains a descriptor data structure. The variable points to the descriptor. If you do an **addc** operation on the



Key:
Symbol Table Class
External Class
Abstract
Instantiated
virtual
non-virtual
linenumber
loader_symbol
member
variable
location
located_symbol
label
block
code_unit
subroutine
soid_reference_symbol
named_constant
pathname_reference_symbol
common
soid_obj
symbol
undiscovered_symbol
file
scope
image
module
symbol_bag
namespace
aggregate_type
enum_type
float_type
int_type
function_type
void_type
intlike_type
error_type
type
stringchar_type
typedef
undiscovered_type
ds_undiscovered_type
array_type
opaque_type
pointer_type
qualified_type
reference_type
char_type
code_type

descriptor, you can then do an **indirect** operation to locate the data.

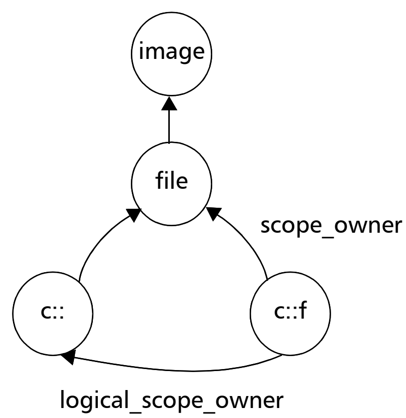


delayed_symbol	Indicates if a symbol has been full or partially read-in. The following constants are or'd and returned: **skim**, **index**, **line**, and **full**.
demangler	The name of demangler used by your compiler.
element_addressing	The location containing additional operands that let you go from the data's base location to an element.
enumerators	Name of the enumerator tags. For example, if you have something like **enum[R,G,B]**, the tags would be **R**, **G**, and **B**.
external_name	When used in data types, it translates the object structure to the type name for the language. For example, if you have a pointer that points to an **int**, the external name is **int** *.
full_pathname	This is the # separated static path to the variable. For example, **##image#file#external-name**... .
id	The internal object handle for the symbol. These symbols always take the form *number*|*number*.
index_type_index	The array type's index **type_index**. For example, this indicates if the index is a 16-, 32-, 64-bit, and so on.
inheritance	For C++ variables, this string is as follows: **[virtual] [{** **private | protected | public** } **]** **[base class]**
is_argument	A true/false value indicating if a variable was a parameter (dummy variable) passed into the function.
kind	One of the symbol types listed in the first column of the previous table.
language	A string containing a value such as C, C++, or Fortran.

length The byte size of the object. For example, this might represent the size of an array or a subroutine.

location The location in memory where an object's storage begins.

logical_scope_owner
The current scope's owner as defined by the language's rules.

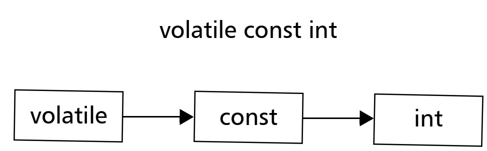


lookup_scope This is a pathname reference symbol that refers to the scope in which to look up a pathname.

lower_bound The location containing the array's lower bound. This is a numeric value, not the location of the first array item.

ordinal The order in which a member or variable occurred within a scope.

qualification A qualifier to a data type such as **const** or **volatile**. These can be chained together if there is more than one qualifier.



resolved_symbol_id
The soid to lookup in a soid reference symbol.

resolved_symbol_pathname
The pathname to lookup in a fortran reference symbol.

return_type_index
The data type of the value returned by a function.

scope_owner The ID of the symbol's scope owner. (This is illustrated by the figure within the **logical_scope_owner** definition.)

static_chain The location of a static link for nested subroutines.

static_chain_height For nested subroutines, this indicates the nesting level.

stride_bound Location of the value indicating an array's stride.

submembers If you have an array of aggregates or pointers and you have already dived on it, this property gives you a list of {*name type*} tuples where **name** is the name of the member of the array (or * if it's an array of pointers), and **type** is the soid of the type that should be used to dive in all into that field.

target_type_index The type of the following entities: **array**, **ds_undiscovered_type**, **pointer**, and **typedef**.

type_index One of the following: **member**, **variable**, or **named_constant**.

upper_bound The location of the value indicating an array's upper bound or extent.

validator The name of an array or pointer validator. This looks at an array descriptor or pointer to determine if it is allocated and associated.

value For enumerators, this indicates the item's value in hexadecimal bytes.

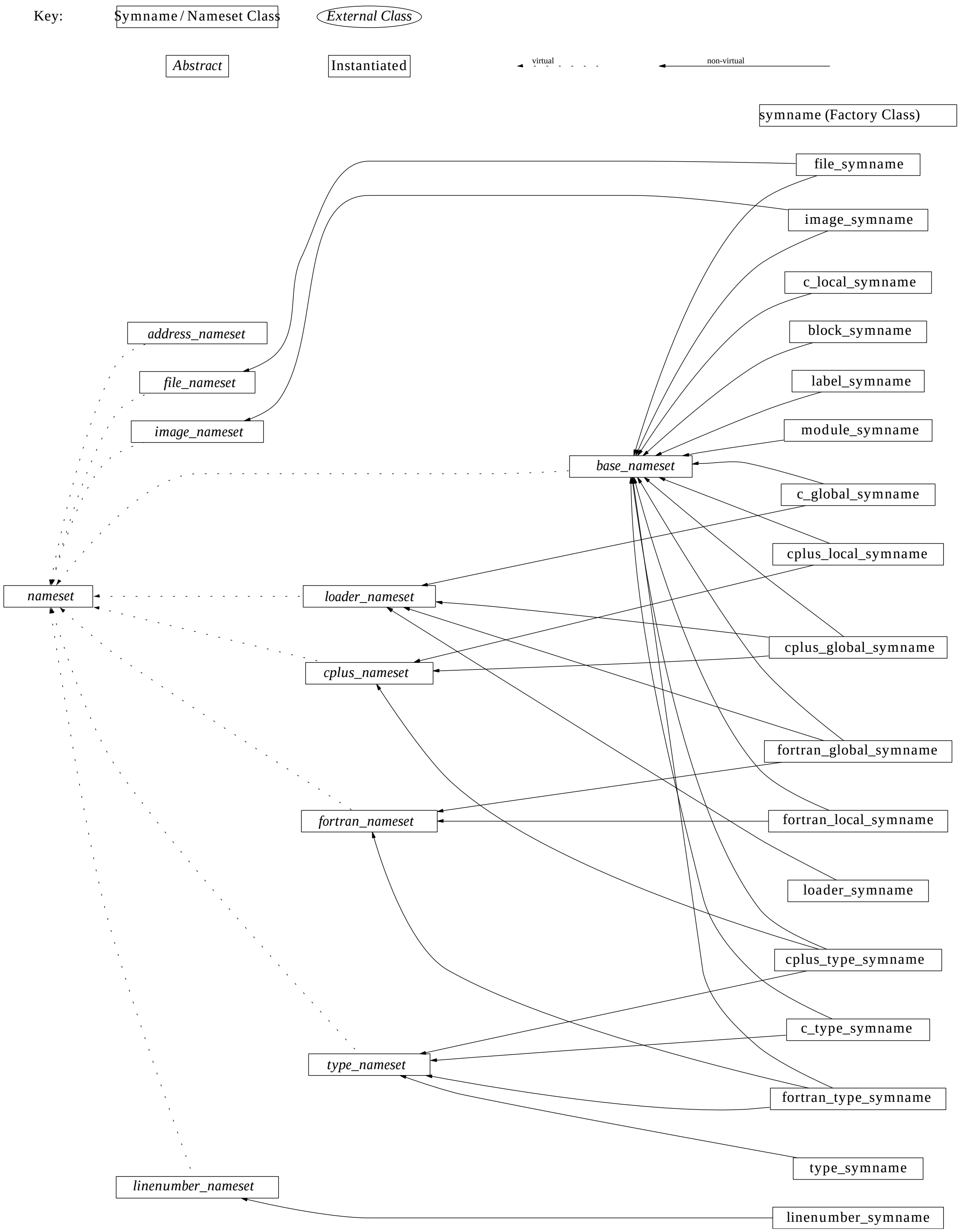
value_size For enumerators, this indicates the length in bytes

Symbol Namespaces

The symbols described in the previous section all reside within namespaces. Like symbols, namespaces also have properties. The figure on the next page illustrates how these namespaces are related.

The following table lists the properties associated with a namespace.

Symbol Namespaces	Properties	
block_symname	base_name	
c_global_symname	base_name	loader_name
	loader_file_path	
c_local_symname	base_name	
c_type_symname	base_name	type_index
cplus_global_symname	base_name	cplus_template_types
	cplus_class_name	cplus_type_name


Key:
Symname / Nameset Class
External Class
Abstract
Instantiated
virtual
non-virtual
symname (Factory Class)
file_symname
image_symname
c_local_symname
block_symname
label_symname
module_symname
c_global_symname
cplus_local_symname
cplus_global_symname
fortran_global_symname
fortran_local_symname
loader_symname
cplus_type_symname
c_type_symname
fortran_type_symname
type_symname
linenumber_symname
address_nameset
file_nameset
image_nameset
base_nameset
nameset
loader_nameset
cplus_nameset
fortran_nameset
type_nameset
linenumber_nameset

Symbol Namespaces	Properties	
	cplus_local_name	loader_file_path
	cplus_overload_list	loader_name
cplus_local_symname	base_name	cplus_overload_list
	cplus_class_name	cplus_template_types
	cplus_local_name	cplus_type_name
cplus_type_symname	base_name	cplus_template_types
	cplus_class_name	cplus_type_name
	cplus_local_name	type_index
	cplus_overload_list	
file_symname	base_name	directory_path
	directory_hint	
fortran_global_symname	base_name	loader_file_path
	fortran_module_name	loader_name
	fortran_parent_function_name	
fortran_local_symname	base_name	
	fortran_parent_function_name	
	fortran_module_name	
fortran_type_symname	base_name	fortran_parent_function_name
	fortran_module_name	type_index
image_symname	base_name	member_name
	directory_path	node_name
label_symname	base_name	
linenumber_symname	linenumber	
loader_symname	loader_file_path	loader_name
module_symname	base_name	
type_symname	type_index	

Many of the following properties are used in more than one namespace. The explanations for these properties will assume a limited context as their use is similar. Some of these definitions assume that you're are looking at the following function prototype:

```
void c::foo<int>(int &)
```

base_name The name of the function. For example, **foo**.

cplus_class_name
The C++ class name. For example, c.

cplus_local_name
Not used.

cplus_overload_list
The function's signature. For example, **int &**.

cplus_template_types
The template used to instantiate the function. For example: **<int>**.

cplus_type_name
The data type of the returned value; for example, *void*.

directory_hint The directory to which you were attached when you started TotalView.

directory_path Your file's pathname as it is named within your program.

fortran_module_name
The name of your module. Typically, this looks like **module‘var** or **module‘subr‘var**.

fortran_parent_function_name
The parent of the subroutine. For example, the parent is **module** in a reference such as **module‘subr**. If you have an inner subroutine, the parent is the outer subroutine.

linenumber The line number at which something occurred.

loader_file_path The file's pathname.

loader_name The mangled name.

member_name In a library, you might have an object reference. For example, **libC.a(foo.so)**. **foo.so** is the member name.

node_name Not used.

type_index A handle that points to the type definition. It's format is **<number,number,number>**.

type

Gets and sets type properties

Format: **TV::type** *action* [*object-id*] [*other-args*]

Arguments:

action — The action to perform, as follows:

- **commands** — Displays the subcommands that you can use. The CLI responds by displaying the four subcommands shown here. Do not use other arguments with this option.
- **get** — Gets the values of one or more type properties. The *other-args* argument can include one or more property names. The CLI returns these values in a list, and places them in the same order as the property names you entered.

  If you use the **–all** option as an *object-id*, the CLI returns a list containing one (sublist) element for each object.
- **properties** — Lists a type's properties. Do not use other arguments with this option.
- **set** — Sets the values of one or more type properties. The *other-args* argument contains paired property names and values.

object-id — An identifier for an object. For example, **1** represents process 1, and **1.1** represents thread 1 in process 1. If you use the **–all** option, the operation is carried out on all objects of this class in the current focus.

other-args — Arguments required by the **get** and **set** subcommands.

Description:

The **TV::type** command lets you examine and set the type properties and states. These states and properties are:

enum_values — For an enumerated type, a list of {**name value**} pairs giving the definition of the enumeration. If you apply this to a non-enumerated type, the CLI returns an empty list.

id — The ID of the object.

image_id — The ID of the image in which this type is defined.

language — The language of the type.

length — The length of the type.

name — The name of the type; for example, **class foo**.

prototype — The ID for the prototype. If the object is not prototyped, the returned value is {}.

rank — (array types only) The rank of the array.

struct_fields (**class**/**struct**/**union** types only). A list of lists giving the description of all the type's fields. Each sublist contains the following fields:

{ *name type_id addressing properties* }

where:

name is the name of the field.

type_id is simply the *type_id* of the field.

addressing contains additional addressing information that points to the base of the field.

properties contains an additional list of properties in the following format:

"[virtual] [public|private|protected] base class"

If no properties apply, this string is null.

If you use **get struct_fields** for a type that is not a **class**, **struct**, or a **union**, the CLI returns an empty list.

target For an array or pointer type, returns the ID of the array member or target of the pointer. If this is not applied to one of these types, the CLI returns an empty list.

type Returns a string describing this type. For example, **signed integer**.

type_values Returns all possible values for the **type** property.

Examples: `TV::type get 1|25 length target`

Finds the length of a type and (assuming it is a pointer or an array type) the target type. The result may look something like:

4 1|12

The following example uses the **TV::type properties** command to obtain the list of properties:

```
d1.<> \
proc print_type {id} {
        foreach p [TV::type   properties] {
            puts [format "%13s  %s" $p [TV::type get $id $p]]
```

```
      }
}
d1.<> print_type 1|6
            enum_values
                     id  1|6
               image_id  1|1
               language  f77
                 length  4
                   name  <integer>
              prototype
                   rank  0
          struct_fields
                 target
                   type  Signed Integer
            type_values  {Array} {Array of charac-
                         ters} {Enumeration}...

d1.<>
```

type_transformation

Creates type transformations and examine properties

Format: **TV::type_transformation** *action* [*object-id*] [*other-args*]

Arguments:

action The action to perform, as follows:

commands Displays the subcommands that you can use. The CLI responds by displaying the subcommands shown here. Do not use additional arguments with this subcommand.

create Creates a new transformation object. The *object-id* argument is not used; *other-args* is **Array**, **List**, **Map**, or **Struct**, indicating the type of transformation being created. You can change a transformation's properties up to the time you install it. After being installed, you can longer change them.

get Gets the values of one or more transformation properties. The *other-args* argument can include one or more property names. The CLI returns these property values in a list whose order is the same as the property names you entered.

If you use the **-all** option as an *object-id*, the CLI returns a list containing one (sublist) element for the object.

properties Displays the properties that the CLI can access. Do not use additional arguments with this option. These properties are discussed later in this section.

set Sets the values of one or more properties. The *other-args* argument consists of pairs of property names and values. The argument pairs that you can set are listed later in this section.

object-id The type transformation ID. This value is returned when you crate a new transformation. For example, **1** represents process 1. If you use the **-all** option, the subcommand is carried out on all objects of this class in the current focus.

other-args Arguments required by **get** and **set** subcommands.

Description: The **TV::type_transformation** command lets you define and examine properties of a type transformation. The states and properties you can set are:

addressing_callback

Names the procedure that locates the address of the start of an array. The call structure for this callback is:

addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

This callback defines a TotalView addressing expression that computes the starting address of an array's first element.

compiler Reserved for future use.

id The type transformation ID returned from a **create** operation.

language The language property specifies source language for the code of the aggregate type (class) to transform. This is always C++.

list_element_count_addressing_callback

Names the procedure that determines the total number of elements in a list. The call structure for this callback is:

list_element_count_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

This callback defines an addressing expression that specifies how to get to the member of the symbol that specifies the number of elements in the list.

If your data structure does not have this element, you still must use this callback. In this case, simply return {**nop**} as the addressing expression and the transformation will count the elements by following all the pointers. This can be very time consuming.

list_element_data_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the data member of a list element. The call structure for this callback is:

list_element_data_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

list_element_next_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the next element of a list. The call structure for this callback is:

list_element_next_addressing_callback *id* where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

list_element_prev_addressing_callback

Names the procedure that defines an addressing expression that specifies how to access the previous element of a list. The call structure for this callback is:

list_element_prev_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

This property is optional. For example, you would not use it in a singly linked list.

list_end_value

Specifies if a list is terminated by NULL or the head of the list. Enter one of the following: **NULL** or **ListHead**

list_first_element_addressing_callback

Names the procedure that defines an addressing expression that specifies how to go from the head element of the list to the first element of the list. It is not always the case that the head element of the list is the first element of the list. The call structure for this callback is:

list_element_first_element_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

list_head_addressing_callback

Names the procedure that defines an addressing expression to obtain the head element of the linked list. The call structure for this callback is:

list_head_addressing_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

lower_bounds_callback

Names the procedure that obtains a lower bound value for the array type being transformed. For C/C++ arrays, this value is always 0. The call structure for this callback is:

lower_bounds_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

name Contains a regular expression that checks to see if a symbol is eligible for type transformation. This regular expression must match the definition of the aggregate type (class) being transformed.

type_callback The **type_callback** property is used in two ways.

(1) When it is used within a list or vector transformation, it names the procedure that determines the type of the list or vector element. The callback procedure takes one parameter, the symbol ID of the symbol that was validated during the callback to the procedure specified by the **validate_callback**. The call structure for this callback is:

type_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

(2) When it is used within a struct transformation, it names the procedure that specifies the data type to be used when displaying the struct.

type_transformation_description

A string containing a description of what is being transformed. For example, you might enter "GNU Vector".

upper_bounds_callback

Names the procedure that defines an addressing expression that computes the extent (number of elements) in an array. The call structure for this callback is:

upper_bounds_callback *id*

where *id* is the symbol ID of the symbol that was validated using the **validate_callback**'s procedure.

validate_callback

Names a procedure that is called when a data type matches the regular expression specified in the **name** property. The call structure for this callback is:

validate_callback *id*

where *id* is the symbol ID of the symbol being validated.

Your callback procedure check the symbol's structure to insure that it should be transformed. While not required, most users will extract symbol information such as its type and its data members while validating the data type. The callback procedure must return a Boolean value, where *true* means the symbol is valid and can be transformed.

Index

U

V

W

X

Y